Seaport – A Portable and Efficient Process-oriented Simulator

Kalyan Perumalla

College of Computing, Georgia Tech
Atlanta, GA 30332-0280

GIT-CC-01-02
March 15, 2001

Abstract

Seaport is a process-oriented simulator. Using Seaport, models can be written using a mixture of event-oriented and process-oriented simulation processes. Seaport has been written with the (seemingly conflicting) goals of simplicity, portability and efficiency. It currently runs on **Intel** processors running **Solaris** or **Linux**, and on **SGI** processors running **Irix**. Portability is aided by the use of standard C++ and the **pthreads** multi-threading library.

Contents

Seaport – A Portable and Efficient Process-oriented Simulator

Kalyan Perumalla

March 15, 2001

1 Introduction

Seaport is a process-oriented simulator. Using Seaport, models can be written using a mixture of event-oriented and process-oriented simulation processes. Seaport has been written with the (conflicting) goals of simplicity, portability and efficiency. It currently runs on **Intel** processors running **Solaris** or **Linux**, and on **SGI** processors running **Irix**. Portability is aided by the use of standard C^{++} and the **pthreads** multi-threading library.

2 Class interfaces

2.1 SimEvent

Simulation processes exchange time-stamped events during simulation. These events are instances of *event types*. All events types in the simulation must be derived classes of the base class **SimEvent**, whose class interface is shown in Figure 1.

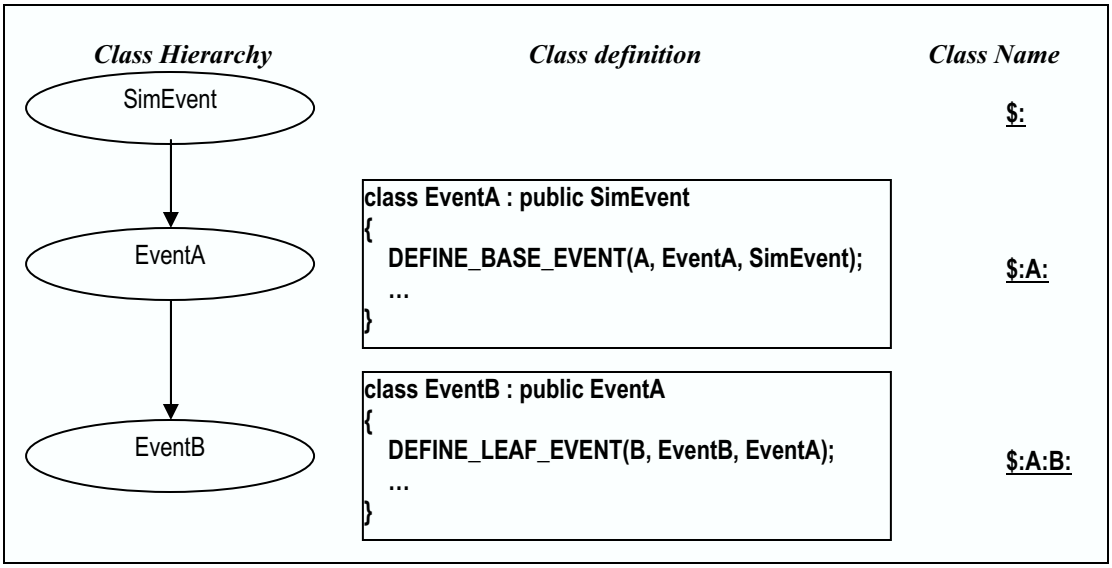
```
class SimEvent
{
   public: SimEvent( void );
   public: SimEvent( const SimEvent &e );
   public: virtual ~SimEvent( void );

   public: virtual const string *name( void ) const;
   public: virtual bool isa( const string &n ) const;
   public: virtual SimEvent *dup( void ) const = 0;

   public: struct { SimProcessID pr; SimFederateID fed; } src, dest;
   public: SimReflectorID rid;
   public: SimTime recv_ts;
   public: SimTime tie_breaker;
   public: long scratch;

   public: void *operator new( size_t sz );
   public: void operator delete( void *d, size_t sz );
};
```

Figure 1 SimEvent **class interface**

2.1.1 Event type naming

Every event type **E** should use either the **DEFINE_BASE_EVENT** or the **DEFINE_LEAFT_EVENT** macro in its class definition. These macros specify the name of all events of type **E** according to the hierarchical event type naming features as described next.

Event types are named based on their position in the class hierarchy, starting from the root base class **SimEvent**. Event names are formed by appending an identifier to the name



of its parent class, followed by a colon ':'. The identifier can be the same as the name of the event class, or it can be any other valid word similar to a *C/C++* identifier. The root base class **SimEvent** has the special name '**$:**'. An event type **EventA** named **A** that is derived from **SimEvent** is named '**$:A:**', and an event type **EventB** named **B** that is derived from **EventA** is named '**$:A:B:**'. Note that the event's *C++* class name and the event name identifier can be different (e.g. **EventA** and **A** respectively).

The macros **DEFINE_BASE_EVENT()** and **DEFINE_LEAF_EVENT()** can be used to define the **name()** method of event types to specify the name identifier for an event. Both the macros take three arguments each: the first argument is the name identifier (e.g. **A**), the second is the *C++* class name of the event type (e.g. **EventA**), and the third argument is the class name of the parent event type from which this event is derived (e.g. **SimEvent**). Events that will eventually be instantiated using the **new** operator should be named as a leaf event, whereas any other event that is simply in the middle of the event hierarchy tree must be defined as a base event. The definition of both the macros is given in Figure 3. The helper macros used in the definition of the convenience macros are shown in Figure 2.

```
#define _EVENT_NAME( _n, _p ) \
   public: virtual const string *name(void) const { static string *s = 0; \
        return s ? s : (s = new string((*_p::name())+#_n+":")); }
#define _EVENT_DUP( _e ) \
   public: virtual SimEvent *dup(void) const { return new _e( *this ); }
```

Figure 2 Helper macros used in event class definition macros

```
#define DEFINE_BASE_EVENT( _n, _e, _p ) _EVENT_NAME( _n, _p )
#define DEFINE_LEAF_EVENT( _n, _e, _p ) _EVENT_NAME( _n, _p ) _EVENT_DUP( _e )
```

Figure 3 Convenience macros used in defining event classes

The **SimEvent::isa()** method can be used to test if an event is of a certain type. With the preceding definitions of **EventA** and **EventB**, the following code illustrates the behavior of **SimEvent::isa()**.

```
EventB *b = new EventB();
cout << b->isa( "$:" );          //prints true
cout << b->isa( "$:A" );         //prints true
cout << b->isa( "$:A:B:" );      //prints true
cout << b->isa( "$:C" );         //prints false
```

Note that for all events, **isa("$:")** always returns true, since all event types are derived from **SimEvent**.

2.1.2 Event copy constructor

Every event type **E** should have a copy constructor **E::E(const E &e)**. This copy constructor is used by the **E::dup()** method to duplicate events of type **E**. The **E::dup()** method is automatically defined as part of the **DEFINE_LEAF_EVENT()** macro which is specified in the type definition of **E**. The copy constructor should perform a deep copy of the event, such that the copy event is independent of the source event.

2.1.3 Event fields

The timestamp of an event is stored in its **recv_ts** member variable, which is the simulation time at which the event is processed by the event's destination(s). The **tie_breaker** field is used to break ties among equal timestamps. The simulation kernel in event list management uses the **scratch** field, and hence that field is not relevant to simulation applications. The **src** and **dest** denote the source and destination simulation processes respectively. The **pr** variable in **src** and **dest** correspond to the IDs of the source and destination processes respectively. The **fed** variable is not utilized currently, and is included for future use. The **rid** value is the ID of the reflector on which this event is sent/received. The rid value is positive if the event is posted to a reflector, and negative if the event is sent directly to a particular destination process.

2.1.4 Event creation and destruction

The **new** and **delete** operators are overloaded for **SimEvent**. These operators are defined to store and reuse event buffers. Last-in-first-out discipline is used for buffer reuse, to improve cache performance. All events are thus allocated and freed by these operators, unless overridden by different overloaded definitions for derived event classes.

2.2 SimProcess

Simulation processes are defined as derived classes of the base class SimProcess. The class interface for the base process class SimProcess is shown in Figure 4.

Processes can be instantiated and added to the simulation at any time before or during the simulation. Processes are added using the **add()** method of the singleton instance of the Simulator class (see Section 2.6). The addition of a process can be immediate or can be scheduled for some future point in simulation time. At the time the process is actually added, the simulator invokes the **init()** method of the process. Events can be scheduled during process initialization. Note that the **init()** method is separate from the process constructor. While the process constructor is executed when the process is instantiated, the **init()** method is executed at the moment the process is actually added to the simulation. Any initialization that cannot be performed in the constructor can be performed during

initialization as part of the **init()** method. The **receive()** or **read()** method of a process is invoked by the simulator whenever the current simulation time reaches the timestamp of an event destined to that process. The **wrapup()** method of the process is invoked just before the process is deleted from the simulation using the **del()** method of the simulator. If a process is not removed from simulation (using **Simulator::del()**) before the process is deleted, then the destructor of **SimProcess** automatically invokes **wrapup()** and deletes the process.

A process can use the **now()** method to obtain the current simulation time.

```
class SimProcess
{
   public: SimProcess( void );
   public: virtual ~SimProcess( void );
   public: virtual void init( void );
   public: virtual void wrapup( void );

   public: virtual SimTime now( void ) const;

   public: virtual SimReflectorID reflector( const string &rname );
   public: virtual void publish( const SimReflectorID &rid, const string &eventname );
   public: virtual void subscribe( const SimReflectorID &rid, const string &eventname );

   public: virtual SimEventID send( SimProcessID to, SimEvent *e, SimTime dt=0,
                           SimTime tie=0, SimFederateID fed=-1 );
   public: virtual SimEventID post( const SimReflectorID &rid,  SimEvent *e, SimTime dt=0, SimTime tie=0 );
   public: virtual SimTimerID set_timer( SimTime dt, SimTime tie = 0 );

   public: virtual void receive( SimEvent *e );
   public: virtual void read( const SimEvent *e );

   public: virtual SimEvent *retract( SimEventID eid );
   public: virtual void retract_timer( SimTimerID tid );
   public: virtual void timedout( SimTimerID timer_id );

   public: SimProcessID ID;
};
```

Figure 4 SimProcess **class interface**

2.2.1 Event exchange

There are two ways by which processes can exchange events: (1) direct event exchange (2) indirect event exchange. In general, the direct method is more efficient than the indirect method with respect to runtime performance, but the indirect method is more flexible and powerful from a software engineering point of view.

Direct event exchange

The direct method is the traditional method in which the sender process of the event designates a unique destination process. The sender process uses the **send()** method for this purpose. The destination process receives the event through its **receive()** method. The destination process is responsible for deleting the event and freeing up its memory.

Since the **send()** method requires the ID of the destination process, the sender process needs to be aware of the destination process and its ID. This implies tight coupling between the sender and destination processes.

Indirect event exchange

In the indirect method, the sender process posts the event to a *reflector* (see Section 2.2.3 for a description of reflectors). Those processes that are subscribed to that reflector receive the event. The sender process uses the **post()** method to send the event, and the destination processes receive the event through their **receive()** or **read()** methods. If there is exactly one process subscribed to the reflector, then the event is given to the process through its **receive()** method. The process is then responsible for deleting the event and freeing up its memory. If there is more than one process subscribed to the reflector, then all but the last process are given a read-only copy of the event through their **read()** method, while the last process is handed over the event through its **receive()** method.

Note that the **read()** method takes a **const** event as argument while the **receive()** method takes a non-const event as its argument. Hence, processes must be prepared to receive events through both **receive()** as well as **read()** methods. In situations where the destination process receives a read-only event through its **read()** method but needs a read-write copy of the event for processing that event, then, the process can use the **dup()** method of the event to make a copy of the event for its exclusive use.

Since the **post()** method only requires a reflector ID, the sender process does not need to be aware of the destination processes. This implies a relaxed relation among sender and destination processes, which can be developed and added to the application independently.

2.2.2 Event retraction

Events that have been sent previously can be retracted later using the **retract()** method, which takes the ID of the scheduled event as its argument and returns the retracted event. It is an error to attempt to retract an event after the simulation time reaches or advances beyond the event's timestamp. When the event is sent, using either **send()** or **post()**, the event's ID is returned by those methods, which can be used as argument to the **retract()** method.

2.2.3 Reflectors

Reflectors are like mailing lists or newsgroups, and processes are like mailing list subscribers and publishers that read and post to the lists. Processes declare their intention to receive events on a reflector by *subscribing* to that reflector. Similarly, processes declare their intention to post events to a reflector by *publishing* to that reflector. All the processes subscribed to a reflector receive the events sent to that reflector. During subscription, processes can specify the set of event types in which they are interested. Only events belonging to the specified types posted to that reflector are seen by those processes. Thus, subscription is based on both reflector and event types. Similarly, processes specify the types of events that they intend to post on a reflector.

The **reflector()** method is used to obtain a handle to the reflector with a given name. Reflector names are ordinary strings. If no reflector with the given name exists, one is

created, and a handle to that reflector is returned. If a reflector with the given name already exists, then the handle to that reflector is returned.

The **publish()** method is used by a process to declare its intention to post events of the given type to a given reflector. Any number of event types can be published on a reflector. It is an error to later post on a reflector an event whose type was not specified in a **publish()** call for that reflector by that process.

The **subscribe()** method is used by a process to declare its interest to receive all events of a given type posted to a given reflector. A process can subscribe to any number of event types on any number of reflectors. Event types can correspond to base or leaf types in the event hierarchy. If a base event type is specified, then all types of events derived from that base event type are matched. Thus, for example, if the process desires to receive all events on a reflector irrespective of their event types, then it can subscribe to event type '**$:**' which corresponds to the root event base class **SimEvent**. Since '$:' matches all events, any event arriving on the reflector gets delivered to the process.

Typically, a process performs all the necessary setup for using reflectors during the process initialization, such as in its **init()** method. All calls to **reflector()**, **publish()** and **subscribe()** are typically performed during initialization. However, it is conceivable for a process to perform the same even after initialization.

2.2.4 Setting and retracting timers

Often, it is necessary for simulation processes to set a timer for a certain period of simulation time. This is achieved using the **set_timer()** primitive, which takes the amount of simulation time **dt** into the future after which the timer is set to expire. The **set_timer()** method sets a timer to expire at simulation time **now()+dt**, and returns the ID of the timer. At the time the timer expires, ties of this timer with any other timers and/or events with same timestamps are resolved using the **tie** argument. A process can set any number of timers for itself, and all the timers are independent of each other. The process for which the timer is set is called the timer's owner process.

Whenever a timer expires (i.e., simulation time advances to the expiry time of the timer), then the **timedout()** method of the owner process is executed. The ID of the timer is given as argument to the **timedout()** method.

It is sometimes necessary to retract a previously scheduled timer. The **retract_timer()** method can be used for this purpose, by passing the ID of the scheduled timer. It is an error to attempt to retract a timer after the simulation time reaches or exceeds the timer's expiry time.

2.3 SimpleSimProcess

Event-oriented processes should be derived from the **SimpleSimProcess**, which is a simple wrapper over the **SimProcess** class. The **SimpleSimProcess** class is intended to shield users from changes to the **SimProcess** class. It is recommended that event-oriented processes be derived from **SimpleSimProcess** class than from **SimProcess**. Currently, **SimpleSimProcess** is defined as a **typedef** to **SimProcess** as follows.

```
typedef SimProcess SimpleSimProcess;
```

2.4 PeriodicSimProcess

The **PeriodicSimProcess** class derived from **SimProcess** is a convenience class that is useful in modeling processes that have a regular periodic behavior. Its class interface is shown in Figure 5. The periodicity of behavior is specified as argument to the constructor. If the process is added to the simulation (using **Simulator::add()**) at time **T**, and the periodicity is **d**, then, the **tick()** method of the derived class will automatically get executed at **T+d**, **T+2d**, **T+3d** and so on, until the process is removed from simulation (using **Simulator::del()**).

```
class PeriodicSimProcess : public SimProcess
{
   public: PeriodicSimProcess( SimTime period );
   public: virtual ~PeriodicSimProcess( void );
   public: virtual void tick( void ) = 0;
};
```

Figure 5 SimpleSimProcess **class interface**

If the derived class provides its own overloaded versions of **init()**, **timedout()** and **wrapup()** methods, then those overloaded versions must call the corresponding methods belonging to the **PeriodicSimProcess**. This is required since the **PeriodicSimProcess** implementation depends on execution of its own versions of those methods.

2.5 ThreadedSimProcess

In process-oriented models, processes retain full stack context across simulation time advances. Such processes are defined using the **ThreadedSimProcess** class as defined in Figure 6.

```
class ThreadedSimProcess : public SimProcess
{
   public: ThreadedSimProcess( void );
   public: virtual ~ThreadedSimProcess( void );

   public: struct WaitContext
   {
      int type; const SimEvent *ce; SimEvent *e;
      WaitContext(int t=0) : type(t), ce(0), e(0) {}
      virtual void reset( void ) { ce = e = 0; }
      virtual void receive( const SimEvent *cx, SimEvent *x ) { ce=cx; e=x; }
   };

   public: virtual void run( void ) = 0;
   public: virtual bool filter( WaitContext *context );
   public: virtual void wait( SimTime dt, WaitContext *wc = 0 );
   public: virtual void wait( WaitContext *wc ) { wait( -1, wc ); }
};
```

Figure 6 ThreadedSimProcess **class interface**

ThreadedSimProcess is implemented using **pthreads**, using one thread per process, hence the name. The main elements of a threaded process are its **run()**, **filter()** and **wait()** methods as described next.

2.5.1 Run, filter and wait

Classes derived from **ThreadedSimProcess** must define a **run()** method, which is the main activity thread of that process. Within the **run()** method, events can be sent at any time. In addition, the **wait()** method can be invoked to suspend the process until a certain condition is satisfied. When the process resumes later, the **wait()** call returns normally, retaining full stack context just as it would if the **wait()** were a simple function call.

Two flavors of **wait()** are provided: (1) with time and context arguments (2) with a single context argument only. The latter is a special case of the former, with a default value for the time argument of the former.

With the two arguments of the **wait(SimTime dt, WaitContext *wc)** method, four cases are possible:

Case 1: **dt >= 0 && wc == 0**: In this case, the **wait()** functions as a pure time-lapse method that waits until the simulation time reaches **now()+dt**.

Case 2: **dt < 0 && wc == 0**: This is an illegal case and not allowed.

Case 3: **dt < 0 && wc != 0**: This is similar to case 4, except that no timeout period is specified.

Case 4: **dt >= 0 && wc != 0**: This is the more general case, in which a timeout period **dt** for waiting is specified along with a wait context **wc**. The wait context **wc** is used in conjunction with the **filter()** method to determine when the **wait()** successfully completes. Whenever an event arrives destined to the waiting process, the simulator automatically performs the following sequence of steps:

i. The wait context is initialized with the received event. This is done by invoking **wc->receive()** with the received event as its argument. The first argument to **receive()** is always the received event. If the received event is read-only, then the second (non-const) argument is passed as null. Otherwise, the second argument is the same as the first argument, which is the received event. In other words, the first argument specifies the event, while the second argument specifies whether that event is read-only or read-write. The default implementation of **receive()** is to assign the received arguments into member variables **ce** and **e**. This behavior can be overridden by defining a derived class of the base class **ThreadedSimProcess::WaitContext** and using instances of the derived class to **wait()**.

ii. The waiting process's **filter()** method is invoked with the wait context **wc** as the argument. The **filter()** method can be used to either filter unwanted events, or buffer events for later processing, or terminate the **wait()** for immediate processing of the received event. The **filter()** method is responsible to determine if the receipt of the event is sufficient for the **wait()** to complete successfully. It must return **true** if the **wait()** must be completed to resume the process; otherwise, it must return **false** to indicate that the process must continue waiting. The **filter()** method or the **WaitContext** must also together ensure that read-write events are deleted properly.

iii. If the **filter()** method returns **true**, then the simulator causes the **wait()** to successfully complete, so that the **run()** method of the process resumes where it

was suspended in the **wait()** call. If the **filter()** method returns **false**, then the simulator calls the **reset()** method on the wait context, and then continues to keep the process suspended.

When the **wait()** is invoked by the process, the simulator internally sets a timer to expire after **now()+dt**. If the timer expires before **filter()** returns true for any event, then the **wait()** completes abnormally, with null values for the wait context event variables **wc->ce** and **wc->e**.

In order to be able to distinguish among different wait contexts in the **filter()** method, wait context variables can set a different type variable using the **type** argument in the **WaitContext** constructor. This is useful if the process suspends itself at multiple points during execution. The **type** value can be used to perform different actions in **filter()** depending on at which point the process is suspended.

2.6 Simulator

The application is responsible for creating an instance of the **Simulator** class. The class interface of **Simulator** is shown in Figure 7.

```
class Simulator
{
   public: Simulator( void );
   public: virtual ~Simulator( void );

   public: virtual void start( void );
   public: virtual SimTime next( SimTime max_t, int max_nevents );
   public: virtual void stop( void );

   public: virtual void add( SimProcess *p, SimTime dt=0 );
   public: virtual void del( SimProcess *p, SimTime dt=0 );

   public: virtual int get_nfeds( void );
   public: virtual SimFederateID get_fedid( void );

   public: static int dbg;
};
```

Figure 7 Simulator **class interface**

The **Simulator** class is implemented as a singleton, with only one instance permitted in the entire application. The constructor of **Simulator** automatically initializes a global variable called **sim** to the created **Simulator** instance. Thus, after an instance of **Simulator** is created, it is globally accessible via the global variable **sim**. The declaration of **sim** is shown in Figure 8.

```
extern Simulator *sim;
```

Figure 8 Global variable pointing to the Simulator **singleton instance**

After the simulator is created, it should be initialized using the **start()** method. After that, the application can give cycles to the simulator to execute simulation events in batches. This is done using the simulator's **next()** method. The amount of simulation progress can be controlled using the arguments to the **next()** method. The **max_t** argument specifies a

limit on the simulation time up to which the simulator can advance the simulation before returning back to the application. The **max_nevents** specifies a limit on the number of events to execute before returning back to the application. The simulator returns from the **next()** method when the earlier of the two conditions is met. The **next()** method returns the timestamp of the next earliest unprocessed event in the event list. If there are no more unprocessed events, then **MAX_DBL** is returned.

The **next()** method can be used in different ways depending on the needs of the application. In the simplest case, the application can transfer control to the simulator until the simulation completes, using the following loop, with any large integral value for **batch**:

```
while( sim->next(MAX_DBL, batch) < MAX_DBL);
```

The application can query the simulator for the next unprocessed event timestamp without executing any events, by invoking **next()** with a value of zero for **max_nevents**.

As described previously, processes can be added to and deleted from the simulation at any time during the simulation. A process is added using the simulator's **add()** method, and removed using the **del()** method. If the **dt** argument is zero, then the process is added immediately to the simulation. Otherwise, the process is scheduled to be added after **dt** simulation time units from the current time (**dt** should not be negative).

Simulation can be stopped by invoking the **stop()** method of the simulator. The **stop()** method forces deletion of all existing simulation processes (as though **Simulator::del()** has been invoked for those processes), and the processes are freed using the C++ delete operator on them. All pending future events are also freed from the event list.

2.7 Miscellaneous

To be completed.

```
typedef double SimTime;
#define MAX_DBL 1e37
```

```
typedef SimEvent *SimEventID;
typedef SimEventID SimTimerID;
typedef long SimProcessID;
typedef short SimFederateID;
typedef long SimReflectorID;
```